

Chapter 5:

Programming the

Intel 8086

Specifics of ALP – Data Definition Directives

- A *variable* is a symbolic name for a location in memory. This is done because it is easy to remember variables, but not memory locations. It is like an aka, or a pseudonym.
-

Data Definition Directives

- Variables are identified by *labels*. A label shows the starting location of a variable's memory location. A variable's offset is the distance from the beginning of the data segment to the beginning of the variable.
-

Data Definition Directives (cont.)

- A label does not indicate the length of the memory that the variable takes up.
 - If a string is being defined, the label offset is the address of the first byte of the string (the first element of the string).
 - The second element is the offset + 1 byte.
 - The third element is offset +2 bytes.
-

Data Definition Directives (cont.)

- The amount of memory to be allocated is determined by the directive itself.

DB	Define byte
DW	Define word
DD	Define doubleword
DF DP	Define far pointer
DQ	Define quadword
DT	Define tenbytes

Define Byte

- Allocates storage for one or more 8-bit values (bytes). Has the following format:
`[name] DB initialvalue [,initialvalue]`
 - The name is the name of the variable. Notice that it is optional.
-

Define Byte (cont)

- initialvalue can be one or more 8-bit numeric values, a string constant, a constant expression or a question mark.
 - If signed, it has a range of 127 to -128, If unsigned, it has a range of 0 - 255.
-

Define Byte - Example

```
char          db    'A'    ; ASCII character
signed1       db    -128   ; min signed value
signed2       db    127    ; max signed value
unsigned1     db    0      ; min unsigned value
unsigned2     db    255    ; max signed value
```

Define Byte (cont)

- Multiple values: A sequence of 8-bit numbers can be used as initialvalue. They are then grouped together under a common label, as in a list.
- The values must be separated by commas.

```
list db 10,20,30,40
```

Define Byte (cont)

- The 10 would be stored at offset 0000;
 - 20 at offset 0001;
 - 30 at offset 0002; and
 - 40 at offset 0003,
 - where 0001 represents a single byte.
-

Define Byte (cont)

- A variable's value may be left undefined. This can be done by placing a '?' for each byte to be allocated (as in a list).

count db ?

Define Byte (cont)

- A string may be assigned to a variable, each of whose elements will be allocated a byte.

```
c_string db 'This is a long string'
```

- The length of a string can be automatically determined by the assembler by the '\$' symbol.
-

Define Word

- Serves to allocate memory to one or more variables that are 16 bits long. Has the following format:
[name] DW initialvalue [,initialvalue]
 - The name is the name of the variable. Notice that it is optional.
 - initialvalue can be one or more 16-bit numeric values, a string constant, a constant expression or a question mark.
-

Define Word (cont)

- If signed, it has a range of 32,767 to -32,768,
 - If unsigned, it has a range of 0 - 65,535.
-

Define Word (Example)

```
var          dw      1,2,3 ; defines 3 words
signed1      dw      -32768 ; smallest signed value
signed2      dw      32767  ; largest signed value
unsigned1    dw      0      ; smallest unsigned value
unsigned2    dw      65535  ; largest signed value
var-bin      dw      1111000011110000b
var-hex      dw      4000h
var-mix      dw      1000h,4096,'AB',0
```

Reverse Storage Format

- The assembler reverses the bytes in a word when storing it in memory.
 - The lowest byte is placed in the lowest address.
 - It is re-reversed when moved to a 16-bit register.

value	dw	2AB6h
B6		2A

Define Doubleword – DD

- Same as DB and DW, except the memory allocated is now 4 bytes (2 words, 32 bits).
[name] DD initialvalue [,initialvalue]
 - The name is the name of the variable. Notice that it is optional.
 - *initialvalue* can be one or more 32-bit numeric values, either in dec., hex or bin. form, string const., a const. Expression, or ?
-

Multiple Values

- A sequence of 32-bit numbers can be used as initialvalue.
 - They are then grouped together under a common label, as in a list.
 - The values must be separated by commas.
-

Reverse Order Format

- As in define word, the bytes in doubleword are stored in reverse order as in DW.
- For example,

```
var      dd  12345678h
          78  56  34  12
```

Type Checking

- When a variable is created, the assembler characterizes it according to its size (i.e., byte, word, doubleword, etc.).
- When a variable is later referenced, the assembler checks its size and only allows values to be stored that come from a register or other memory that matches in size.
- ~~Mismatched movements of data not allowed.~~

Data Transfer Instructions - mov

- The instruction mov is called the data transfer instruction.
 - A very important one in assembly - much programming involves moving data around.
 - Operands are 8- or 16-bit on the 8086, 80186 and 80286.
 - Operands on the 80386 and beyond, they can also be 32-bits long.
-

Data Transfer Instructions –

mov (cont)

- The format is:

mov *destination, source*

- The source and destination operands are self-explanatory.
 - The source can be an immediate value (a constant), a register, or a memory location (variable). It is not changed by the operation.
 - The destination can be a register or a memory location (variable).
-

Operands

- Register Operands: Transfer involving only registers. It is the fastest.
 - Source = any register
 - Destination = any register except CS and IP
 - Immediate Operands
 - Immediate value can be moved to a register (all but IP) or to memory.
 - Destination must be of same type as source.
-

Operands (cont.)

- Direct operands
 - A variable may be one of the two operands, but not both. It does not matter which is the variable.
-

Limitations on operands

- There are some limitations on mov:
 - CS or IP not destination operand
 - Moving immediate data to segment registers.
 - Moving from segment register to segment register.
 - Source and destination operands of different types.
 - Immediate value as destination (obviously!!)
 - Memory to memory moves
-

Sequential Memory Allocation

- Memory for variables is allocated sequentially by the assembler.
- If we call DB several times, such as in:

```
var1    db    10  
var2    db    15  
var3    db    20
```

Sequential Memory Allocation (cont)

- `var1` will be the first byte in the data segment of main memory.
 - This segment may be identified by the base segment and the offset.
 - `var2` will occupy the next available memory location, or 1 byte away from the beginning of the data segment in memory.
-

Sequential Memory Allocation (cont)

- var 3 will be 2 bytes away from this starting point.
- This will be the case even if the memory locations are not labeled, such as in:

db 10

db 20

db 30

Offsets

- Many times, memory will be allocated, but not labeled.
 - This is typical of an array, when only the entire array is labeled, not each cell.
 - The address of the array is the address of the first element (position) of the array.
 - All subsequent cells are allocated by adding an offset to the address of the head element.
-

Offsets

- This is also true when a list of elements is defined through DB, DW, or DD.
 - Example: an array or list of 8-bit numbers whose memory location is called *a-list*.
 - To access the first element of a-list, we reference the location in memory corresponding to a-list.
 - To access any of the other elements of the array, we provide an offset to the address of array.
 - The second element at array+1, the third at array+2, the fourth at array+3, etc.
-

Offsets

- To move the value of the 5th element of the array to register AL:

```
mov    al , array+4
```

- The size of the two operands must match.
 - Otherwise, an error may result.
 - Note that AL is used, not AX - 1 byte.
-

PTR Operator

- Used to clarify an operand's type. NOT a pointer.
- It can be placed between the **mov** command and the operands, as for example: Used for readability only. It does not change the size of the operand in any way.

```
mov word ptr count,10  
mov byte ptr var2,5
```

XCHG Instruction

- Allows the direct exchange of values between 2 registers or between a register and a memory location.
 - Very fast
 - Needs to obey size constraints
 - Used in sorting.
-

Stack Operations

- Already discussed what a stack is.
 - Each position in the stack is 2 bytes long
– only 16-bit registers can be copied into the stack.
 - The “bottom” of the stack is in high memory, and it grows downward.
 - Typically, 256 bytes are allocated to the stack, enough for 128 entries.
-

Stack Operations (cont)

- Identified by the SS register (stack segment), which identifies the address of the base location of the stack segment.
 - The stack pointer (SP register) indicates the address of the first element of the stack (top of the stack).
-

Push Operation

- Puts something (a 16-bit element) at the top position of the stack.
 - Decrements stack pointer (it grows downward).
 - Can put the contents of a register or of memory (a variable)
 - In 80286 and later processors, it can also place an immediate value.
-

Push Operation

- Has the following form:

push ax

push ar1

push 1000h



Pop Operation

- The opposite of the push operation.
 - Removes the top element in the stack.
 - Copies value of top element in stack to destination indicated in the statement.
 - Increments stack pointer.
 - Registers CS (code segment) and IP (instruction pointer) cannot be used as operands.
-

Pop Operation

- Has the following form:

pop ax

pop ar2

PUSHF and POPF

- Special instructions that move and remove the contents of the Flags register onto and out of the stack.
 - These are used to preserve the contents of these registers in case it is changed and the old values are to be reinstated.
-

PUSHA (80186+) and PUSHHD (80386+)

- Pushes the contents of the registers AX, CX, DX, BX, original SP, BP, SI, and DI on the stack in this exact order.
 - PUSHHD does the same for 32-bit registers.
-

POPA and POPD

- Pops the same registers in the reverse order.
-

Load-Effective Address

- LEA : loads a 16- or 32-bit reg. with the offset address of the data specified by the operand
 - Table 4-9
 - 1st ex. : $AX \leftarrow \text{operand offset address NUMB}$
 - LEA BX,[DI] : loads offset address specified by [DI] (contents of DI) into BX
 - MOV BX,[DI] : loads the data stored at the memory location addressed by [DI] into BX
 - OFFSET directive : performs the same function as an LEA instruction if operand is a displacement
-

TABLE 4-9 Load-effective address instructions.

<i>Assembly Language</i>	<i>Operation</i>
LEA AX,NUMB	Loads AX with the address of NUMB
LEA EAX,NUMB	Loads EAX with the address of NUMB
LDS DI,LIST	Loads DS and DI with the 32-bit contents of data segment memory location LIST
LDS EDI,LIST	Loads DS and EDI with the 48-bit contents of data segment memory location LIST
LES BX,CAT	Loads ES and BX with the 32-bit contents of data segment memory location CAT
LFS DI,DATA1	Loads FS and DI with the 32-bit contents of data segment memory location DATA1
LGS SI,DATA5	Loads GS and SI with the 32-bit contents of data segment memory location DATA5
LSS SP,MEM	Loads SS and SP with the 32-bit contents of memory location MEM

- EX. : loads SI with address of DATA1 and DI with address of DATA2, exchanges the contents of these memory location

```

0000          .MODEL SMALL          ;select SMALL model
0000          .DATA                 ;start of DATA segment
0000  2000    DATA1    DW    2000H    ;define DATA1
0002  3000    DATA2    DW    3000H    ;define DATA2
0000          .CODE                 ;start of CODE segment
          .STARTUP                 ;start of program
0017  BE 0000 R    LEA    SI,DATA1    ;address DATA1 with SI
001A  BF 0002 R    MOV    DI,OFFSET DATA2 ;address DATA2 with DI

001D  8B 1C        MOV    BX,[SI]      ;exchange DATA1 with DATA2
001F  8B 0D        MOV    CX,[DI]
0021  89 0C        MOV    [SI],CX
0023  89 1D        MOV    [DI],BX

          .EXIT                     ;exit to DOS
          END                       ;end of file

```

LEA

- LEA and MOV with OFFSET : 3 bytes instruction
- why is LEA available if OFFSET accomplishes the same task ?

1. OFFSET only functions with simple operands such as LIST(not used [DI], LIST[DI], and so on)

2. OFFSET is more efficient for simple operands

- longer to execute LEA BX, LIST(2 clocks) than MOV BX, OFFSET LIST(1 clock, 80486)

- because assembler calculates offset

LEA

- MOV BX, DI performs in less time and is often preferred to LEA BX, [DI]

3. LEA SI,[BX+DI] : modulo-64K sum

- drops any carry out of the 16-bit result
- Ex. BX = 1000H, DI = FF00H : SI = 0F00H

LDS, LES, LFS, LGS, LSS

- load any 16-bit or 32-bit register with an offset address, and the DS, ES, FS, GS, or SS segment register with a segment address
- use any of memory-addressing modes to access a 32-bit(16-bit offset, segment address) or 48-bit(32-bit offset, 16-bit segment address) section of memory that contains both the segment and offset address
- may not use the register addressing(MOD=11)
- LES, LGS, LSS : 80386~

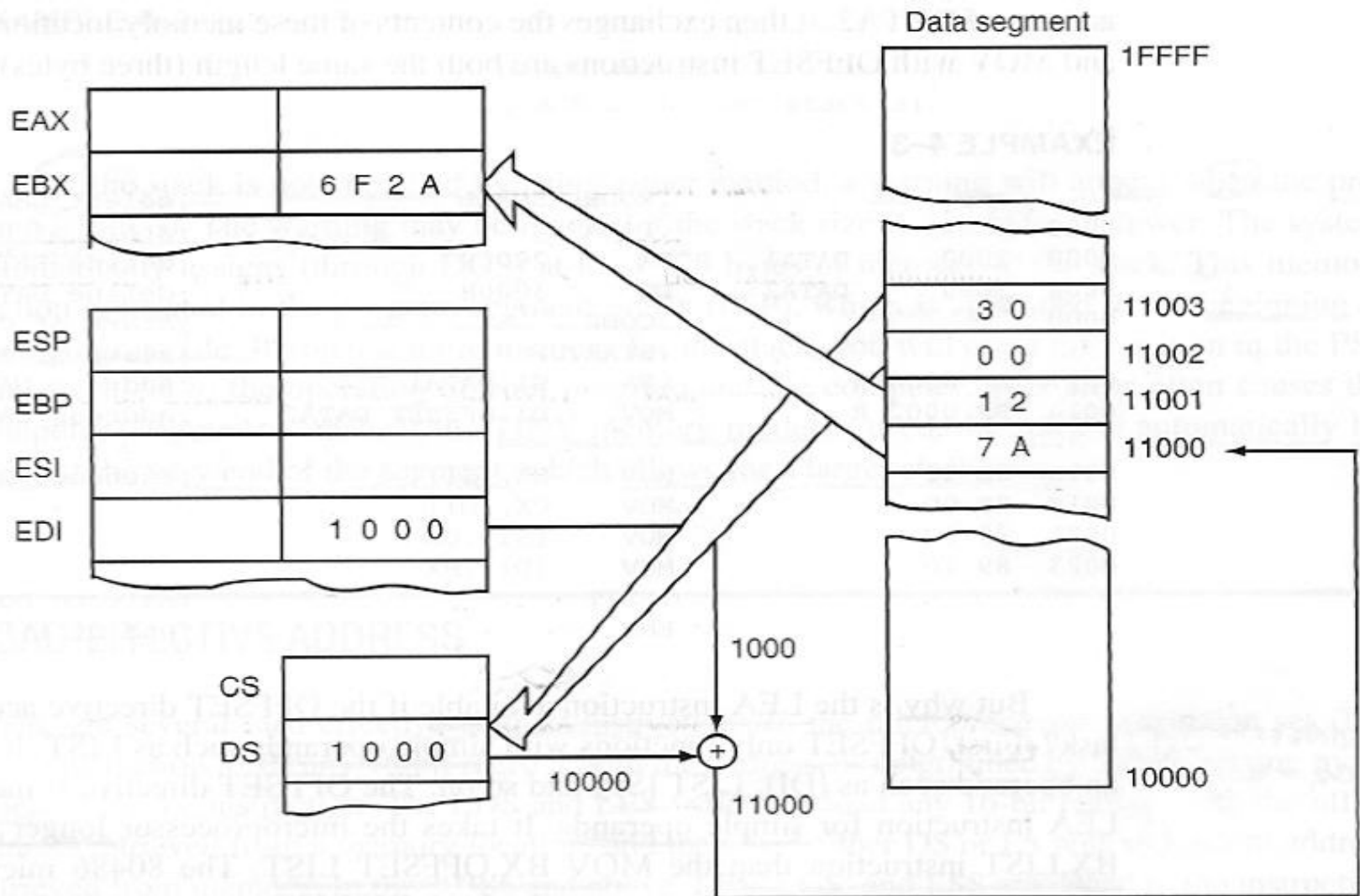


FIGURE 4-15 The `LDS BX,[DI]` instruction loads register `BX` from addresses 11000H and 11001H and register `DS` from locations 11002H and 11003H. This instruction is shown at the point just before `DS` changes to 3000H and `BX` changes to 127AH.

LDS, LES, LFS, LGS, LSS

- obtain a new far address from memory
- far address(offset, segment address) can be stored in memory by the assembler
- ADDR DD FAR PTR FROG : stores far address of FROG in 32-bits of memory at location ADDR
- LDS EBX,[DI] in 80386~ : loads EBX from 4-byte section of memory addressed by DI in the data segment, and following this 4-byte offset is a word that is loaded to DS
- EX: creates a new stack area after

Example

```

                                .MODEL SMALL           ;select small model
                                .386                 ;select 80386
0000                                .DATA             ;start data segment
0000 00000000          SADDR DD      ?              ;old stack address
0004 1000 [          SAREA DW      1000H DUP(?)      ;new stack area
                                ;????
                                ]

2004 = 2004          STOP EQU THIS WORD            ;define top of new stack
0000                                .CODE             ;start code segment
                                .STARTUP             ;start program
0010 FA              CLI                          ;disable interrupts
0011 8B C4           MOV AX,SP                     ;save old SP
0013 A3 0000 R       MOV WORD PTR SADDR,AX
0016 8C D0           MOV AX,SS                     ;save old SS
0018 A3 0002 R       MOV WORD PTR SADDR+2,AX

001B 8C D8           MOV AX,DS                     ;load new SS
001D 8E D0           MOV SS,AX
001F B8 2004 R       MOV AX,OFFSET STOP            ;load new SP
0022 8B E0           MOV SP,AX
0024 FB             STI                          ;enable interrupts

0025 8B C0           MOV AX,AX                     ;do some dummy instructions
0027 8B C0           MOV AX,AX
0029 9F B2 26 0000 R LSS SP,SADDR                 ;get old stack
                                .EXIT               ;exit to DOS
                                END                 ;end program listing

```

String Data Transfers

- LODS, STOS, MOVS, INS, OUTS
- data transfers that either a single byte, word, or doubleword(or if repeated, a block of bytes, words, doublewords)
- before string instruction, the operation of D flag-bit(direction), DI, and SI must be understood
- D flag selects auto-increment($D = 0$) or auto-decrement($D = 1$) operation for DI, SI registers during string instruction.
 - CLD : clears D flag($D = 0$)
 - STD : sets D flag($D = 1$)

String Data Transfers

- whenever a string instruction transfers a byte, word, doubleword : $DI/SI \leftarrow DI/SI \pm 1, \pm 2, \pm 4$
 - Only actual registers used by string instruction increment or decrement
- DI offset address : accesses data in extra segment for all string instructions that use it
- SI offset address : accesses data, by default, in data segment
- SI Segment assignment : may be changed with a segment override prefix
- DI Segment assignment : is always extra segment

String Data Transfers

- LODS $AL \leftarrow DS:[SI]$
- STOS $ES:[DI] \leftarrow AL$
- MOVS $ES:[DI] \leftarrow DS:[SI]$
- SCAS $AL - ES:[DI]$
- CMPS $DS:[SI] - ES:[DI]$
- INS $ES:[DI] \leftarrow [DX]$
- OUTS $[DX] \leftarrow DS:[SI]$

LODS

- $\text{LODS} : \text{AL}, \text{AX}, \text{EAX} \leftarrow \text{DS}:[\text{SI}]$
 - loads AL, AX, EAX with data stored at the data segment offset address indexed by SI
 - $\text{AL}, \text{AX}, \text{EAX} : \text{SI} \leftarrow \text{SI} \pm 1, \pm 2, \pm 4$
 - $D = 0$: auto-increment, $D = 1$ auto-decrement
 - Table 4-10 : LODSB, LODSW, LODSD
 - operands : are often defined as bytes with DB, as words with DW, as doublewords with DD
-
- Fig. 4-16 : LODSW

Table: Forms of the LODS instruction

<i>Assembly Language</i>	<i>Operation</i>
LODSB	$AL = DS:[SI]; SI = SI \pm 1$
LODSW	$AX = DS:[SI]; SI = SI \pm 2$
LODSD	$EAX = DS:[SI]; SI = SI \pm 4$
LODSQ	$RAX = [RSI]; RSI = RSI \pm 8$ (64-bit mode)
LODS LIST	$AL = DS:[SI]; SI = SI \pm 1$ (if LIST is a byte)
LODS DATA1	$AX = DS:[SI]; SI = SI \pm 2$ (if DATA1 is a word)
LODS FROG	$EAX = DS:[SI]; SI = SI \pm 4$ (if FROG is a doubleword)

Note: The segment register can be overridden with a segment override prefix as in LODS ES:DATA4.

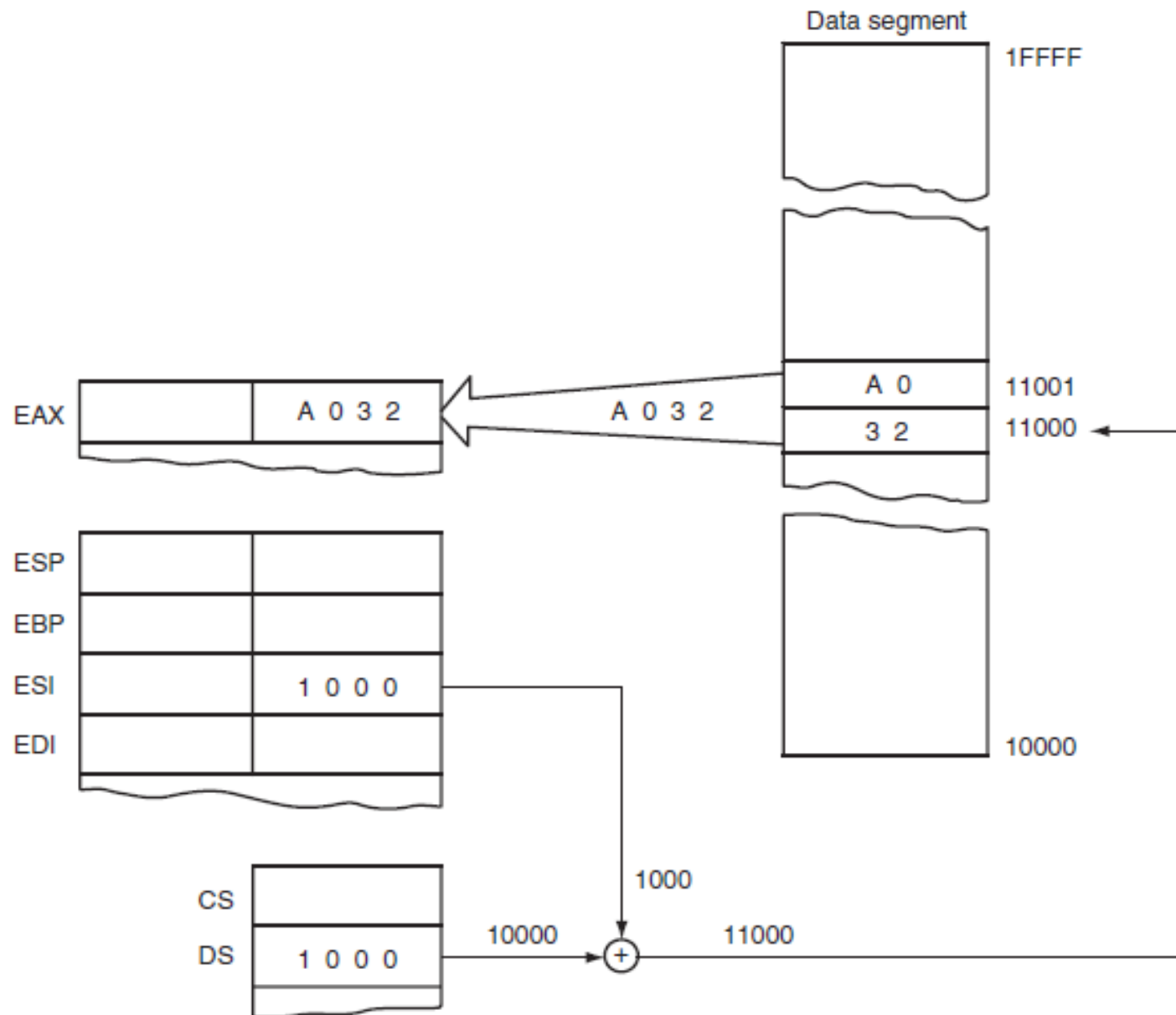


FIGURE 4-18 The operation of the LODSW instruction if `DS = 1000H`, `D = 0`, `11000H = 32`, and `11001H = A0`. This instruction is shown after `AX` is loaded from memory, but before `SI` increments by 2.

STOS

- $\text{STOS} : \text{ES}:[\text{DI}] \leftarrow \text{AL}, \text{AX}, \text{EAX}$
 - stores AL, AX, EAX at the extra segment memory location addressed by DI
 - $\text{AL}, \text{AX}, \text{EAX} : \text{DI} \leftarrow \text{DI} \pm 1, \pm 2, \pm 4$
 - $\text{D} = 0$: auto-increment, $\text{D} = 1$ auto-decrement
 - Table: STOSB, STOSW, STOSD
 - operands : are often defined as bytes with DB, as words with DW, as doublewords with DD
-

- Table: Forms of STOS instruction

<i>Assembly Language</i>	<i>Operation</i>
STOSB	ES:[DI] = AL; DI = DI $\pm$ 1
STOSW	ES:[DI] = AX; DI = DI $\pm$ 2
STOSD	ES:[DI] = EAX; DI = DI $\pm$ 4
STOSQ	[RDI] = RAX; RDI = RDI $\pm$ 8 (64-bit mode)
STOS LIST	ES:[DI] = AL; DI = DI $\pm$ 1 (if LIST is a byte)
STOS DATA3	ES:[DI] = AX; DI = DI $\pm$ 2 (if DATA3 is a word)
STOS DATA4	ES:[DI] = EAX; DI = DI $\pm$ 4 (if DATA4 is a doubleword)

STOS with a REP

- repeat prefix(REP) : is added to any string data transfer instruction, except the LODS
- REP : causes CX to decrement by 1 each time string instruction executes
 - after CX decrements, if $CX \neq 0$, string instruction repeats.
 - if $CX = 0$, instruction terminates and continues with next sequential instruction
- EX. 4-5 : to clear the video text display
 - video text memory : B800:0000(25 line $\times$ 80 character)
 - each character position : 2 bytes(ASCII, color attributes)

EXAMPLE 4-5

		.MODEL TINY		;select TINY model
0000		.CODE		;start of CODE segment
		.STARTUP		;start of program
0100	FC	CLD		;select increment mode
0101	B8 B800	MOV	AX,0B800H	;address segment B800
0104	8E C0	MOV	ES,AX	
0106	BF 0000	MOV	DI,0	;address offset 0000
0109	B9 07D0	MOV	CX,25*80	;load count
010C	B8 0720	MOV	AX,0720H	;load data
010F	F3/AB	REP	STOSW	;clear the screen
		.EXIT		;exit to DOS
		END		;end of file

Arithmetic Instructions

- Form the heart of any program, at any level of abstraction.
 - Integer arithmetic done in 8- or 16-bit operands.
-

Arithmetic Instructions

- Floating point arithmetic done in one of three ways:
 - 80x87 math coprocessor
 - software routines that emulate the coprocessor
 - software that converts floating point to integer, executes the instruction and then converts back to floating point. (not particularly useful in many cases).
 - Section concentrates on integer arithmetic
-

INC and DEC Instructions

- Respectively increment and decrement the operand by one.
- Form is as follows:

```
inc    al    ; incr. 8-bit register by 1
inc    bx    ; incr. 16-bit register by 1
inc    mem   ; incr. memory location by 1
dec    mem   ; decr. memory location by 1
dec    bx    ; decr. 16-bit register by 1
dec    al    ; decr. 8-bit register by 1
```

INC and DEC Instructions

- Faster than ADD and SUB instructions
 - All status flags are affected except the Carry Flag.
-

ADD Instruction

- Used to add two numbers together.

- Format is as follows:

add *destination, source*

- The high level language equivalent of:
destination = destination + source
-

ADD Instruction

- Takes 2 operands, a *source* and a *destination*.
 - Adds the contents of the operands.
 - Places the result in the destination.
 - Only one memory operand may be used.
 - Source may be immediate value, but not destination.
-

ADD Instruction (cont.)

- The contents of the source remain unchanged.
 - A segment register may not be a destination.
 - All status flags are affected.
 - Sizes of operands must match (same size).
-

SUB Instruction

- Subtracts a source operand from a destination operand and stores the result in the destination.
 - Format is as follows:
$$\text{sub } \textit{destination}, \textit{source}$$
 - The high level language equivalent of:
$$\text{destination} = \text{destination} + (-\text{source})$$
-

SUB Instruction (cont.)

- Takes the two's complement of source and adds it to the destination.
 - Only one of the operands may be a memory operand.
 - Size of operands must match (same size).
 - Segment register may not be the destination operand.
-

SUB Instruction (cont.)

- Only one of the operands may be a memory operand.
 - Immediate values cannot be destinations.
-

Effects of SUB and ADD on Flag Registers

- Why the flags?
 - If an operation such as ADD overflows (i.e., number to be put in the destination exceeds the size of the destination), then *overflow*.
 - The flag may indicate that the value of the destination is meaningless.
 - The ADD instruction, therefore, affects both the Carry and Overflow flags.
 - Only 1 of these flags (if signed or if unsigned).
-

Flag Registers (cont.)

- **Zero Flag:** Set when the result of the instructions `inc`, `dec`, `add` or `sub` = 0.
 - **Sign Flag:** Set when the result of the instructions `inc`, `dec`, `add` or `sub` < 0
 - **Carry Flag:** Used with unsigned arithmetic only, even though the processor updates it even if operation is signed.
 - For example:
-

Flag Example

```
mov    ax,00FFh
```

```
add    al,1      ; AX = 0000h, CF=1
```

```
00FFh + 0001h = 0100h
```

- Since the operand of the add instruction is 8-bit (AL register), this is an 8-bit operation.
- Therefore, the operation looks more like

$$\text{FFh} + \text{01h} = \text{100h}$$

Flag Example (cont.)

- This is an overflow, so the Carry Flag is set to indicate that the result was larger than the 8-bit destination could store.
- Could be fixed by using a 16-bit operation (e.g.,)

```
add    ax,1    ; now it's a 16-bit operation
                ; AX = 0100h CF = 0
```

Flag Example (cont.)

- If result of operation is too large for the destination operand, the Carry flag is set.
- Can also occur in a subtraction operation when subtracting a larger operand from a smaller one (signed results are not permissible).

```
mov    ax,5501h
```

```
sub     al,2      ; AX = 55FFh  CF=1
```

- We subtracted 2 from 1, resulting in a negative number in an unsigned operation.
-

Flag Registers (cont.)

- **Overflow Flag:** Set by the processor regardless of the type of operation, but important only when operation is signed.
 - Signed results are useless when this flag is set
 - Can result when using add or sub
 - But not inc or dec.
-

Flag Registers - Example

- If we add $126 + 2 = 128$, in an 8-bit operation, this will represent an overflow.
 - This operation in binary numbers:
 $01111110 + 00000010 = 10000000$
 - Equivalent to the two's complement of -128, which is completely incorrect, therefore meaningless in signed integers.
 - Therefore, $OF = 1$.
-

Flag Registers - Example

(cont.)

• In the case of subtraction, it gets a little more complicated.

- If we subtract 2 from -128:
 - $-128 - 2 = -130$ which is an overflow, as it does not fit in an 8-bit destination.
 - In binary, this looks like this:
$$\begin{array}{r} 10000000 \text{ (-128 in two's complement)} + \\ 11111110 \text{ (-2 in twos' comp)} \\ \hline = 1 \ 01111110 \end{array}$$
-

Flag Registers - Example (cont.)

- The first 1, of course, is a ninth digit, which does not fit into the 8-bit register.
- Thus, the operation looks like it resulted in

01111110

- which is -126.
 - So, the OF = 1.
-

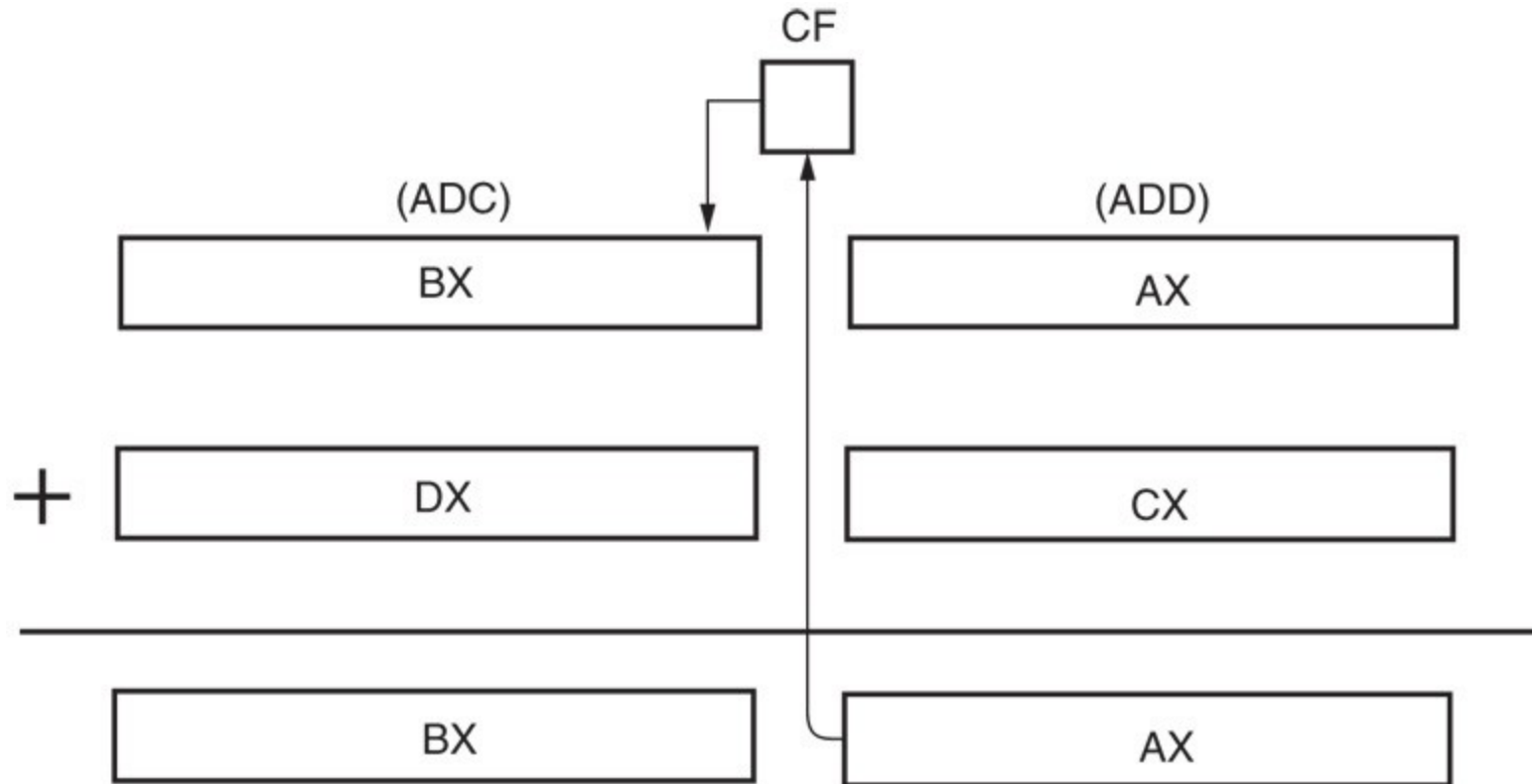
Addition

- Addition (ADD) appears in many forms in the microprocessor.
 - A second form of addition, called **add-with-carry**, is introduced with the ADC instruction.
 - The only types of addition *not* allowed are memory-to-memory and segment register.
 - segment registers can only be moved, pushed, or popped
-

Addition-with-Carry

- ADC adds the bit in the carry flag (C) to the operand data.
 - mainly appears in software that adds numbers wider than 16 or 32 bits in the 80386–Core2
 - like ADD, ADC affects the flags after the addition
 - Figure illustrates this so placement and function of the carry flag can be understood.
 - cannot be easily performed without adding the carry flag bit because the 8086–80286 only
-

Figure Addition-with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.



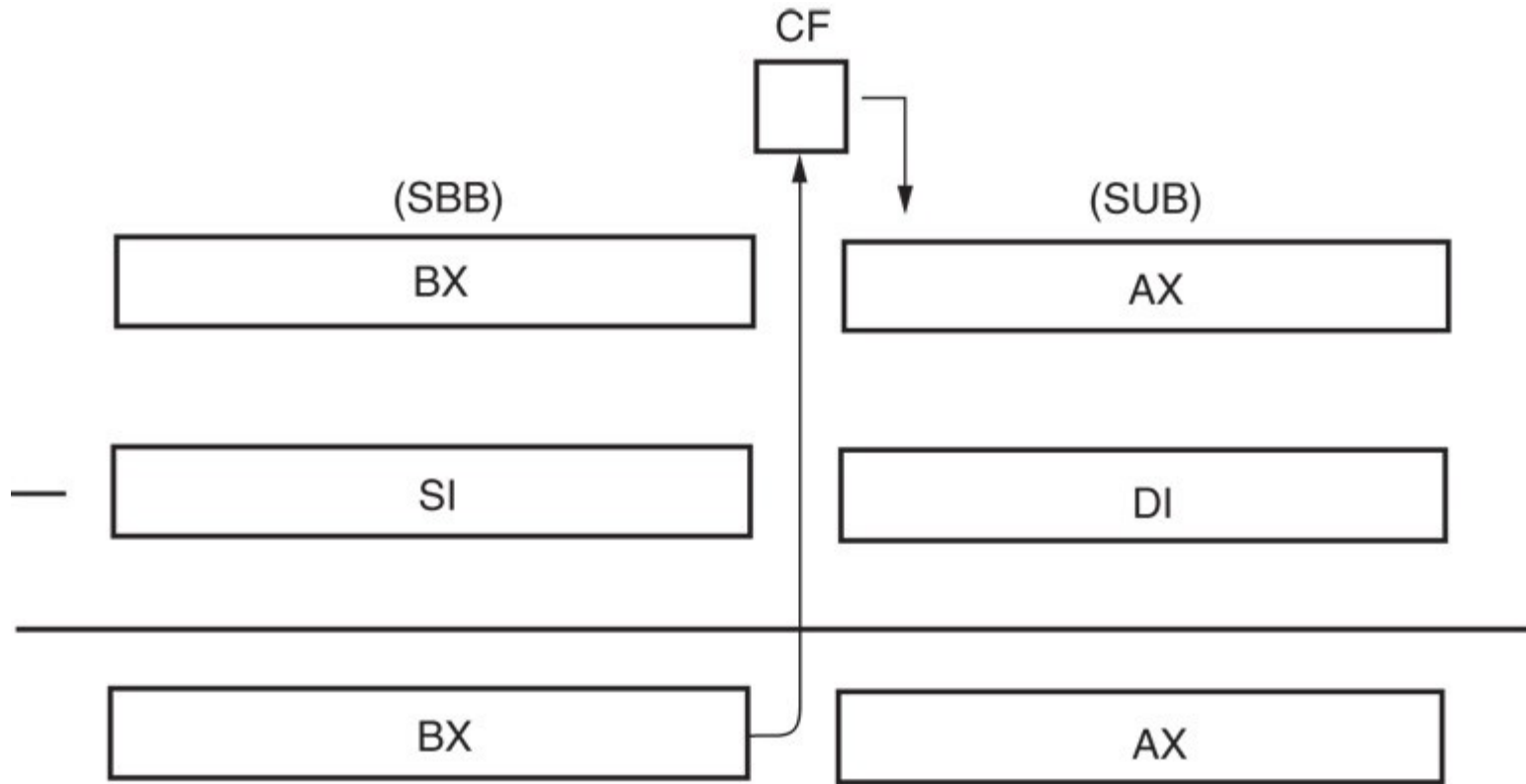
Subtraction

- Many forms of subtraction (SUB) appear in the instruction set.
 - these use any addressing mode with 8-, 16-, or 32-bit data
 - Numbers that are wider than 16 bits or 32 bits must occasionally be subtracted.
 - the **subtract-with-borrow instruction** (SBB) performs this type of subtraction
-

Subtraction-with-Borrow

- A subtraction-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference.
 - most common use is subtractions wider than 16 bits in the 8086–80286 microprocessors or wider than 32 bits in the 80386–Core2.
 - wide subtractions require borrows to propagate through the subtraction, just as wide additions propagate the carry
-

Figure Subtraction-with-borrow showing how the carry flag (C) propagates the borrow.



Comparison

- The comparison instruction (CMP) is a subtraction that changes only the flag bits.
 - destination operand never changes
 - Useful for checking the contents of a register or a memory location against another value.
 - A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.
-

MULTIPLICATION AND DIVISION

- Earlier 8-bit microprocessors could not multiply or divide without the use of a program that multiplied or divided by using a series of shifts and additions or subtractions.
 - manufacturers were aware of this inadequacy, they incorporated multiplication and division into the instruction sets of newer microprocessors.
 - Pentium-Core2 contains special circuitry to do multiplication in as few as one clocking period.
-

8-Bit Multiplication

- With 8-bit multiplication, the multiplicand is always in the AL register, signed or unsigned.
 - multiplier can be any 8-bit register or memory location
- Immediate multiplication is not allowed unless the special signed immediate multiplication instruction appears in a program.
- The multiplication instruction contains ~~one operand because it always~~ multiplies the operand times the

Multiplication

- Performed on bytes, words, or doublewords,
 - can be signed (IMUL) or unsigned integer (MUL)
 - Product after a multiplication always a double-width product.
 - two 8-bit numbers multiplied generate a 16-bit product; two 16-bit numbers generate a 32-bit;
 - two 32-bit numbers generate a 64-bit product
 - in 64-bit mode of Pentium 4, two 64-bit numbers are multiplied to generate a 128
-

16-Bit Multiplication

- Word multiplication is very similar to byte multiplication.
- AX contains the multiplicand instead of AL.
 - 32-bit product appears in DX–AX instead of AX
- The DX register always contains the most significant 16 bits of the product; AX contains the least significant 16 bits.
- As with 8-bit multiplication, the choice of the multiplier is up to the programmer.

Division

- Occurs on 8- or 16-bit and 32-bit numbers depending on microprocessor.
 - signed (IDIV) or unsigned (DIV) integers
 - Dividend is always a double-width dividend, divided by the operand.
 - There is no immediate division instruction available to any microprocessor.
 - In 64-bit mode Pentium 4 & Core2, divide a 128-bit number by a 64-bit number.
-

- A division can result in two types of errors:
 - attempt to divide by zero
 - other is a divide overflow, which occurs when a small number divides into a large number
 - In either case, the microprocessor generates an interrupt if a divide error occurs.
 - In most systems, a divide error interrupt displays an error message on the video screen.
-

8-Bit Division

- Uses AX to store the dividend divided by the contents of any 8-bit register or memory location.
 - Quotient moves into AL after the division with AH containing a whole number remainder.
 - quotient is positive or negative; remainder always assumes sign of the dividend; always an integer
-

- Numbers usually 8 bits wide in 8-bit division .
 - the dividend must be converted to a 16-bit wide number in AX ; accomplished differently for signed and unsigned numbers
 - CBW (**convert byte to word**) instruction performs this conversion.
 - In 80386 through Core2, MOVSX sign-extends a number.
-

16-Bit Division

- Sixteen-bit division is similar to 8-bit division
 - instead of dividing into AX, the 16-bit number is divided into DX-AX, a 32-bit dividend
- As with 8-bit division, numbers must often be converted to the proper form for the dividend.
 - if a 16-bit unsigned number is placed in AX, DX must be cleared to zero
- In the 80386 and above, the number is zero-extended by using the MOVZX

The Remainder

- Could be used to round the quotient or dropped to truncate the quotient.
 - If division is unsigned, rounding requires the remainder be compared with half the divisor to decide whether to round up the quotient
 - The remainder could also be converted to a fractional remainder.
-

Program Control Instructions: Introduction

- This section explains the program control instructions, including the jumps, calls, returns, and interrupts.
-

THE JUMP GROUP

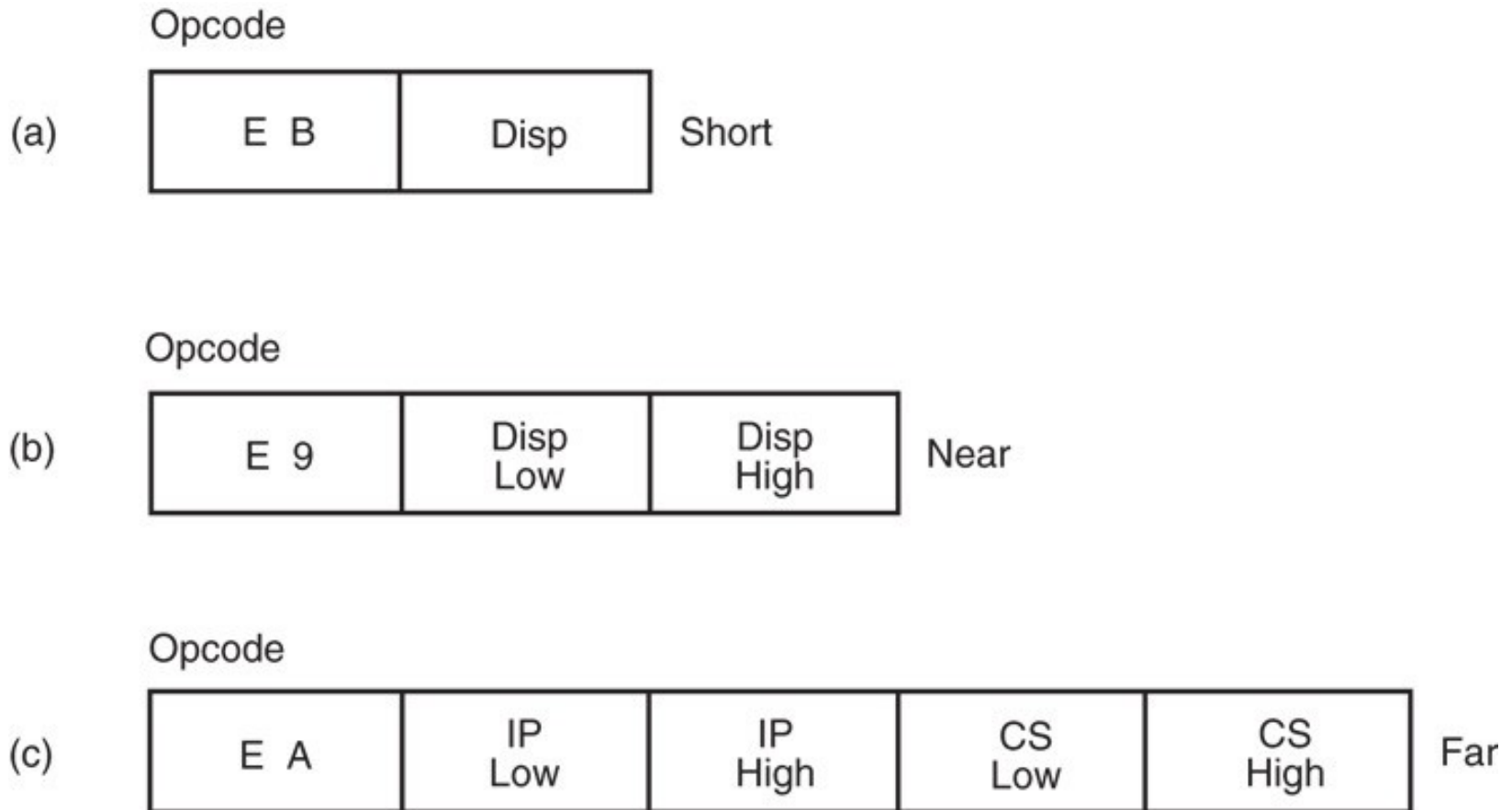
- Allows programmer to skip program sections and branch to any part of memory for the next instruction.
 - A conditional jump instruction allows decisions based upon numerical tests.
 - results are held in the flag bits, then tested by conditional jump instructions
 - LOOP and conditional LOOP are also forms of the jump instruction.
-

Unconditional Jump (JMP)

- Three types: short jump, near jump, far jump.
 - **Short jump** is a 2-byte instruction that allows jumps or branches to memory locations within +127 and -128 bytes.
 - from the address following the jump
 - 3-byte **near jump** allows a branch or jump within $\pm 32\text{K}$ bytes from the instruction in the current code segment.
-

- 5-byte **far jump** allows a jump to any memory location within the real memory system.
 - The short and near jumps are often called **intra-segment jumps**.
 - Far jumps are called **inter-segment jumps**.
-

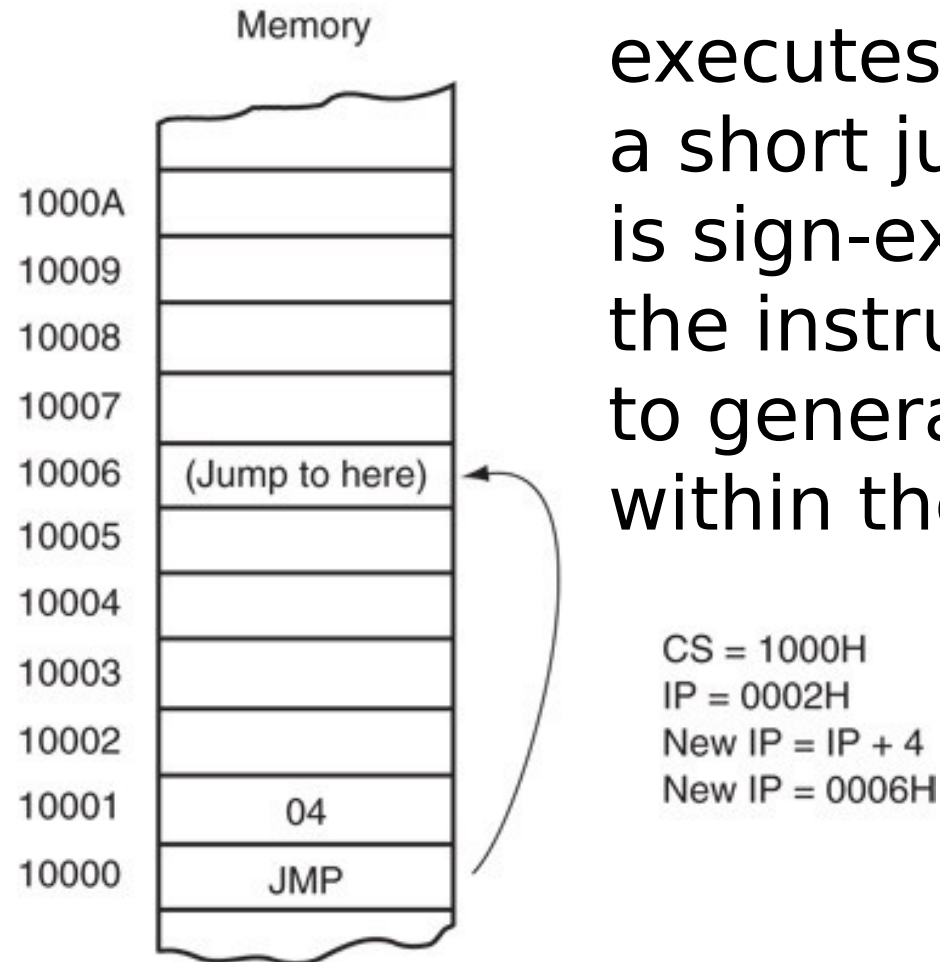
Figure 5-1 The three main forms of the JMP instruction. Note that Disp is either an 8- or 16-bit signed displacement or distance.



Short Jump

- Called **relative jumps** because they can be moved, with related software, to any location in the current code segment without a change.
 - jump address is not stored with the opcode
 - a **distance**, or displacement, follows the opcode
- The short jump displacement is a distance represented by a 1-byte signed number whose value ranges between +127 and -128.
- Short jump instruction appears in Figure

Figure 5-2 A short jump to four memory locations beyond the address of the next instruction.



CS = 1000H
IP = 0002H
New IP = IP + 4
New IP = 0006H

- when the microprocessor executes a short jump, the displacement is sign-extended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment
- The instruction branches to this new address for the next instruction in the program

- When a jump references an address, a label normally identifies the address.
 - The JMP NEXT instruction is an example.
 - it jumps to label NEXT for the next instruction
 - very rare to use an actual hexadecimal address with any jump instruction
 - The label NEXT must be followed by a colon (NEXT:) to allow an instruction to reference it
 - if a colon does not follow, you cannot jump to it
-
- The only time a colon is used is when

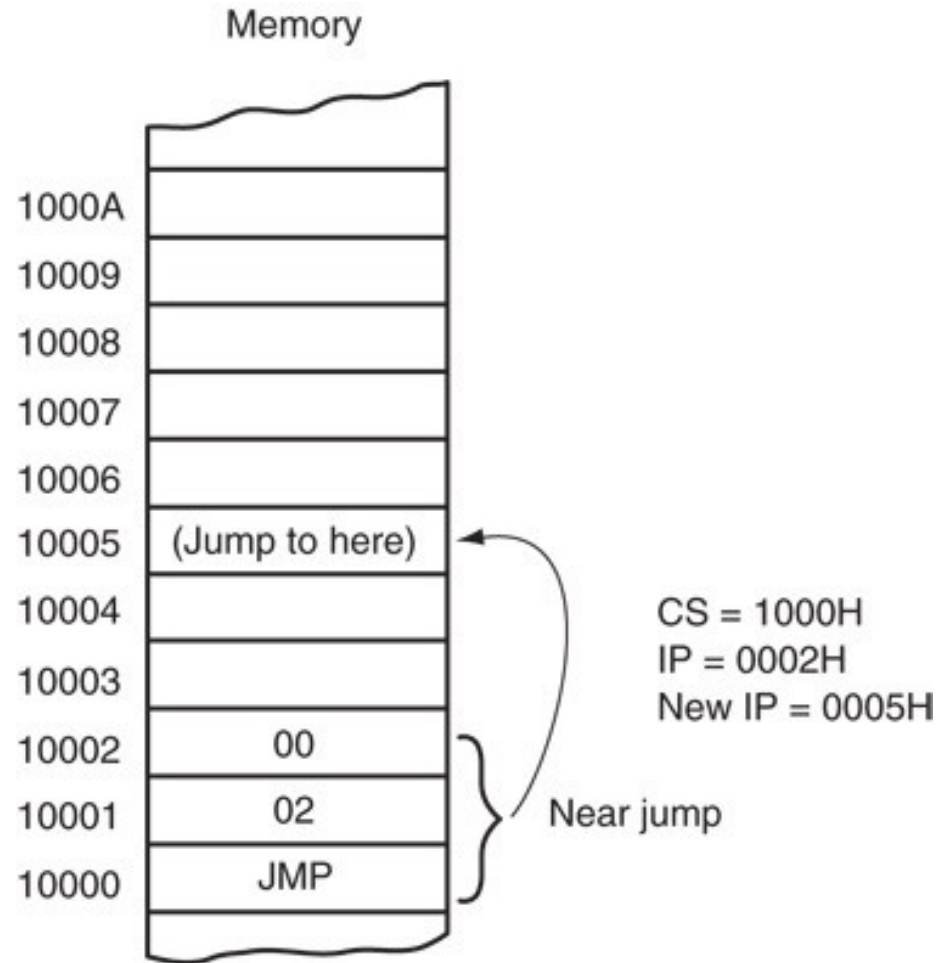
Near Jump

- A near jump passes control to an instruction in the current code segment located within $\pm 32\text{K}$ bytes from the near jump instruction.
 - distance is $\pm 2\text{G}$ in 80386 and above when operated in protected mode
- Near jump is a 3-byte instruction with opcode followed by a signed 16-bit displacement.
 - 80386 - Pentium 4 displacement is 32 bits and

the near jump is 5 bytes long

- Signed displacement adds to the instruction pointer (IP) to generate the jump address.
 - because signed displacement is $\pm 32\text{K}$, a near jump can jump to any memory location within the current real mode code segment
- The protected mode code segment in the 80386 and above can be 4G bytes long.
 - 32-bit displacement allows a near jump to any location within $\pm 2\text{G}$ bytes
- ~~Figure 5-3 illustrates the operation of the real mode near jump instruction~~

Figure 5-3 A near jump that adds the displacement (0002H) to the contents of IP.

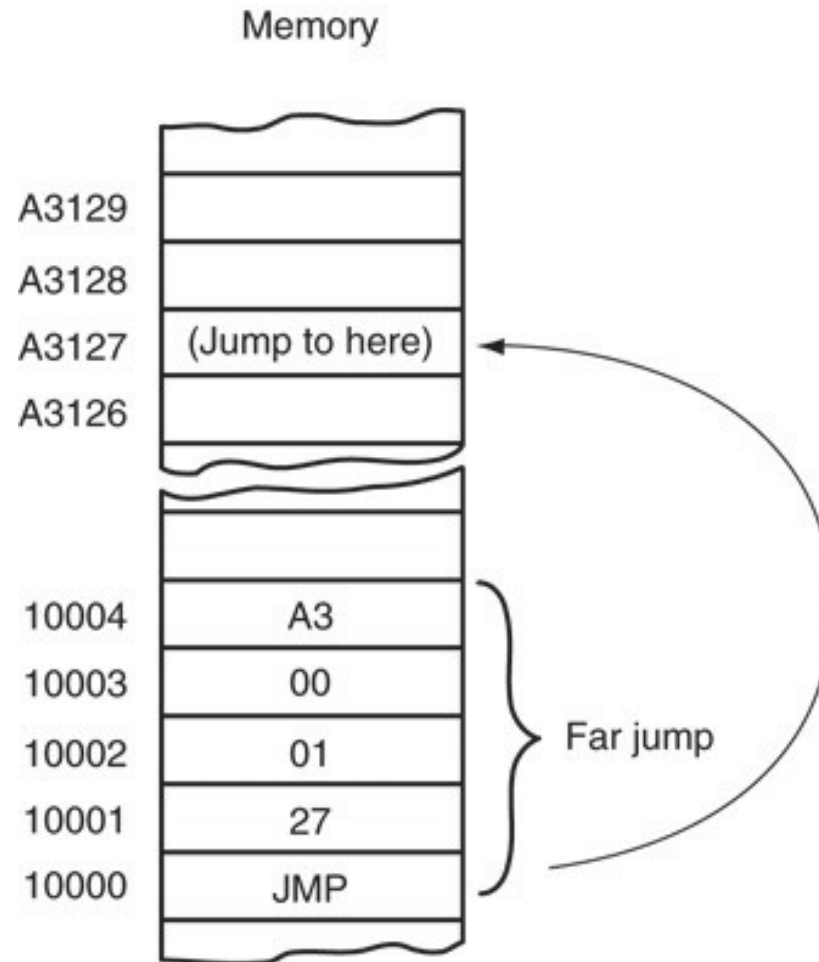


- The near jump is also relocatable because it is also a relative jump.
 - This feature, along with the relocatable data segments, Intel microprocessors ideal for use in a general-purpose computer system.
 - Software can be written and loaded anywhere in the memory and function without modification because of the relative jumps and relocatable data segments.
-

Far Jump

- Obtains a new segment and offset address to accomplish the jump:
 - bytes 2 and 3 of this 5-byte instruction contain the new offset address
 - bytes 4 and 5 contain the new segment address
 - in protected mode, the segment address accesses a descriptor with the base address of the far jump segment
 - offset address, either 16 or 32 bits, contains the offset address within the new code

Figure 5-4 A far jump instruction replaces the contents of both CS and IP with 4 bytes following the opcode.



- The far jump instruction sometimes appears with the FAR PTR directive.
 - another way to obtain a far jump is to define a label as a **far label**
 - a label is far only if it is external to the current code segment or procedure
 - The JMP UP instruction references a far label.
 - label UP is defined as a far label by the EXTRN UP:FAR directive
 - **External labels** appear in programs that contain more than one program file.
-

Jumps with Register Operands

- Jump can also use a 16- or 32-bit register as an operand.
 - automatically sets up as an **indirect jump**
 - address of the jump is in the register specified by the jump instruction
- Unlike displacement associated with the near jump, register contents are transferred directly into the instruction pointer.
- ~~An indirect jump does not add to the instruction pointer.~~

- JMP AX, for example, copies the contents of the AX register into the IP.
 - allows a jump to any location within the current code segment
 - In 80386 and above, JMP EAX also jumps to any location within the current code segment;
 - in protected mode the code segment can be 4G bytes long, so a 32-bit offset address is needed
-

Conditional Jumps

- Always short jumps in 8086 - 80286.
 - limits range to within +127 and -128 bytes from the location following the conditional jump
- In 80386 and above, conditional jumps are either short or near jumps ($\pm 32K$).
 - in 64-bit mode of the Pentium 4, the near jump distance is $\pm 2G$ for the conditional jumps
- ~~Allows a conditional jump to any location within the current code segment.~~

- Conditional jump instructions test flag bits:
 - sign (S), zero (Z), carry (C)
 - parity (P), overflow (O)
 - If the condition under test is true, a branch to the label associated with the jump instruction occurs.
 - if false, next sequential step in program executes
 - for example, a JC will jump if the carry bit is set
 - Most conditional jump instructions are straightforward as they often test one flag bit.
-

- Because both signed and unsigned numbers are used in programming.
 - Because the order of these numbers is different, there are two sets of conditional jump instructions for magnitude comparisons.
 - 16- and 32-bit numbers follow the same order as 8-bit numbers, except that they are larger.
 - Figure 5-5 shows the order of both signed and unsigned 8-bit numbers.
-

Figure 5-5 Signed and unsigned numbers follow different orders.

Unsigned numbers	
255	FFH
254	FEH
...	
132	84H
131	83H
130	82H
129	81H
128	80H
...	
4	04H
3	03H
2	02H
1	01H
0	00H

Signed numbers	
+127	7FH
+126	7EH
...	
+2	02H
+1	01H
+0	00H
-1	FFH
-2	FEH
...	
-124	84H
-125	83H
-126	82H
-127	81H
-128	80H

- When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions.
 - terms *greater than* and *less than* refer to signed numbers
 - When unsigned numbers are compared, use the JA, JB, JAE, JBE, JE, and JNE instructions.
 - terms *above* and *below* refer to unsigned numbers
 - Remaining conditional jumps test individual flag bits, such as overflow and parity.
-

- Remaining conditional jumps test individual flag bits, such as overflow and parity.
 - notice that JE has an alternative opcode JZ
 - All instructions have alternates, but many aren't used in programming because they don't usually fit the condition under test.
-

LOOP

- A combination of a decrement CX and the JNZ conditional jump.
 - In 8086 - 80286 LOOP decrements CX.
 - if $CX \neq 0$, it jumps to the address indicated by the label
 - If CX becomes 0, the next sequential instruction executes
 - In 80386 and above, LOOP decrements either CX or ECX, depending upon instruction mode.
-

- In 16-bit instruction mode, LOOP uses CX; in the 32-bit mode, LOOP uses ECX.
 - default is changed by the LOOPW (using CX) and LOOPD (using ECX) instructions
 - 80386 - Core2
 - In 64-bit mode, the loop counter is in RCX.
 - and is 64 bits wide
 - There is no direct move from segment register to segment register instruction.
-

Conditional LOOPS

- LOOP instruction also has conditional forms: LOOPE and LOOPNE
- LOOPE (**loop while equal**) instruction jumps if CX != 0 while an equal condition exists.
 - will exit loop if the condition is not equal or the CX register decrements to 0
- LOOPNE (**loop while not equal**) jumps if CX != 0 while a not-equal condition exists.

 - will exit loop if the condition is equal or the

PROCEDURES

- A procedure is a group of instructions that usually performs one task.
 - subroutine, method, or **function** is an important part of any system's architecture
 - A procedure is a reusable section of the software stored in memory once, used as often as necessary.
 - saves memory space and makes it easier to develop software
-

- Disadvantage of procedure is time it takes the computer to link to, and return from it.
 - CALL links to the procedure; the RET (**return**) instruction returns from the procedure
 - CALL pushes the address of the instruction following the CALL (**return address**) on the stack.
 - the stack stores the return address when a procedure is called during a program
 - RET instruction removes an address from the stack so the program returns to the instruction following the CALL
-

- A procedure begins with the PROC directive and ends with the ENDP directive.
 - each directive appears with the procedure name
- PROC is followed by the type of procedure:
 - NEAR or FAR
- Procedures that are to be used by all software (global) should be written as far procedures.
- Procedures that are used by a given task (local) are normally defined as near

CALL

- Transfers the flow of the program to the procedure.
 - CALL instruction differs from the jump instruction because a CALL saves a return address on the stack.
 - The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.
-

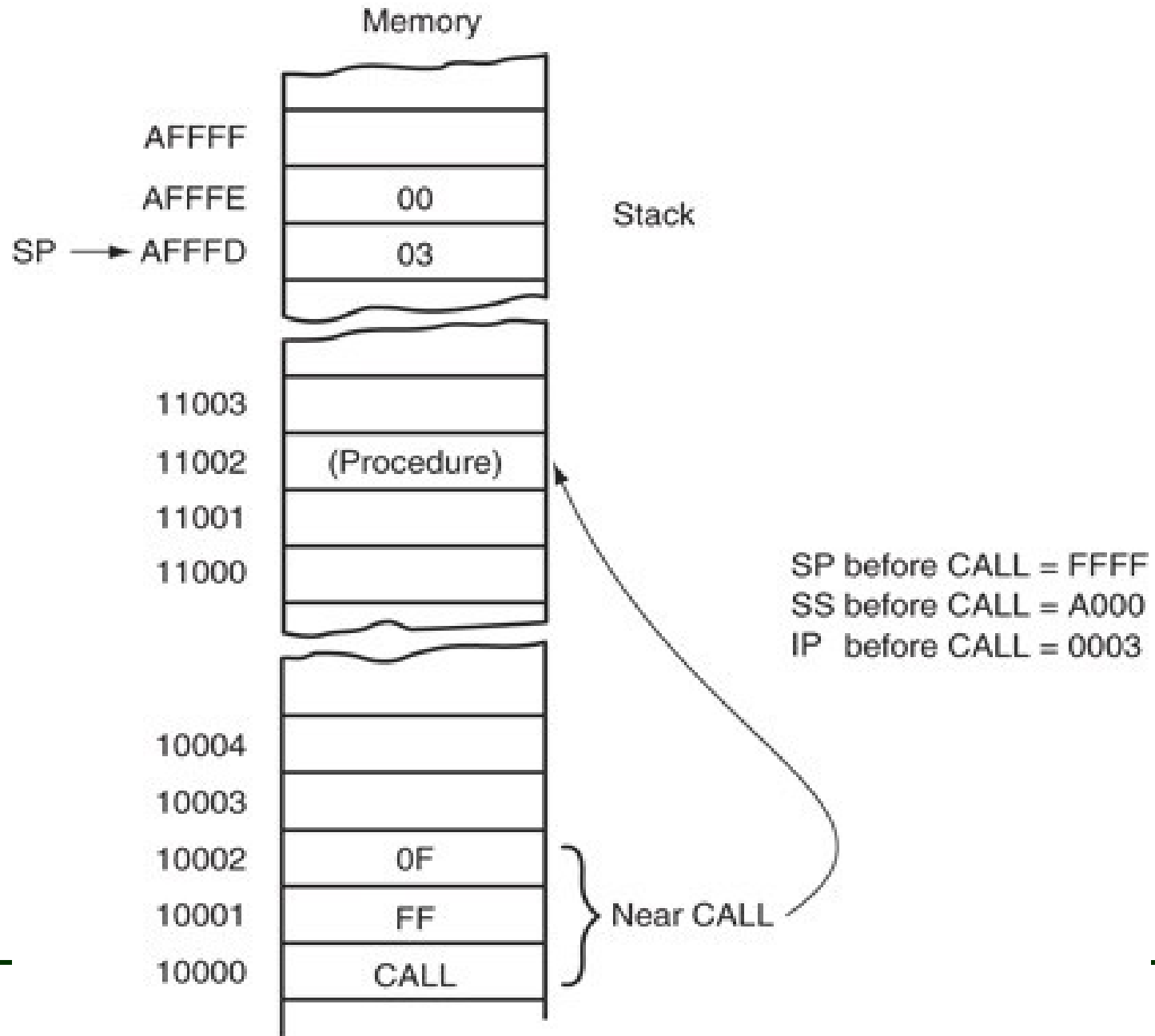
Near CALL

- 3 bytes long.
 - the first byte contains the opcode; the second and third bytes contain the displacement
- When the near CALL executes, it first pushes the offset address of the next instruction onto the stack.
 - offset address of the next instruction appears in the instruction pointer (IP or EIP)
- It then adds displacement from bytes 2 & 3

to the IP to transfer control to the

- Why save the IP or EIP on the stack?
 - the instruction pointer always points to the next instruction in the program
 - For the CALL instruction, the contents of IP/EIP are pushed onto the stack.
 - program control passes to the instruction following the CALL after a procedure ends
 - Figure 5–6 shows the return address (IP) stored on the stack and the call to the procedure.
-

Figure 5-6 The effect of a near CALL on the stack and the instruction pointer.

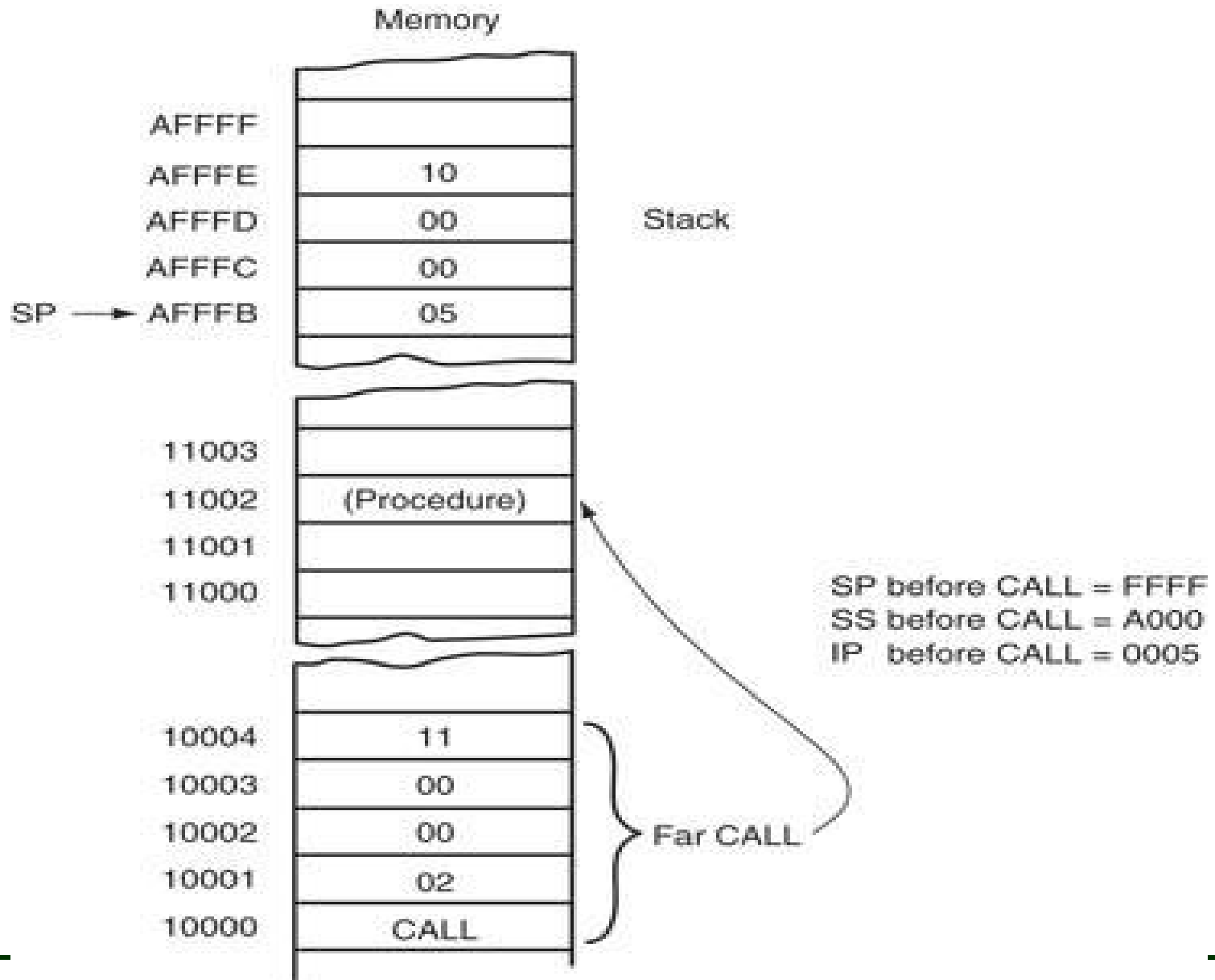


Far CALL

- 5-byte instruction contains an opcode followed by the next value for the IP and CS registers.
 - bytes 2 and 3 contain new contents of the IP
 - bytes 4 and 5 contain the new contents for CS
- Far CALL places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2 through 5.
- ~~This allows far CALL to call a procedure located anywhere in the memory and~~

- Figure 5-7 shows how far CALL calls a far procedure.
 - contents of IP and CS are pushed onto the stack
- The program branches to the procedure.
 - A variant of far call exists as CALLF, but should be avoided in favor of defining the type of call instruction with the PROC statement
- In 64-bit mode a far call is to any memory location and information placed onto the stack is an 8-byte number.
 - the far return instruction retrieves an 8-byte return address from the stack and places it

Figure 5-7 The effect of a far CALL instruction.



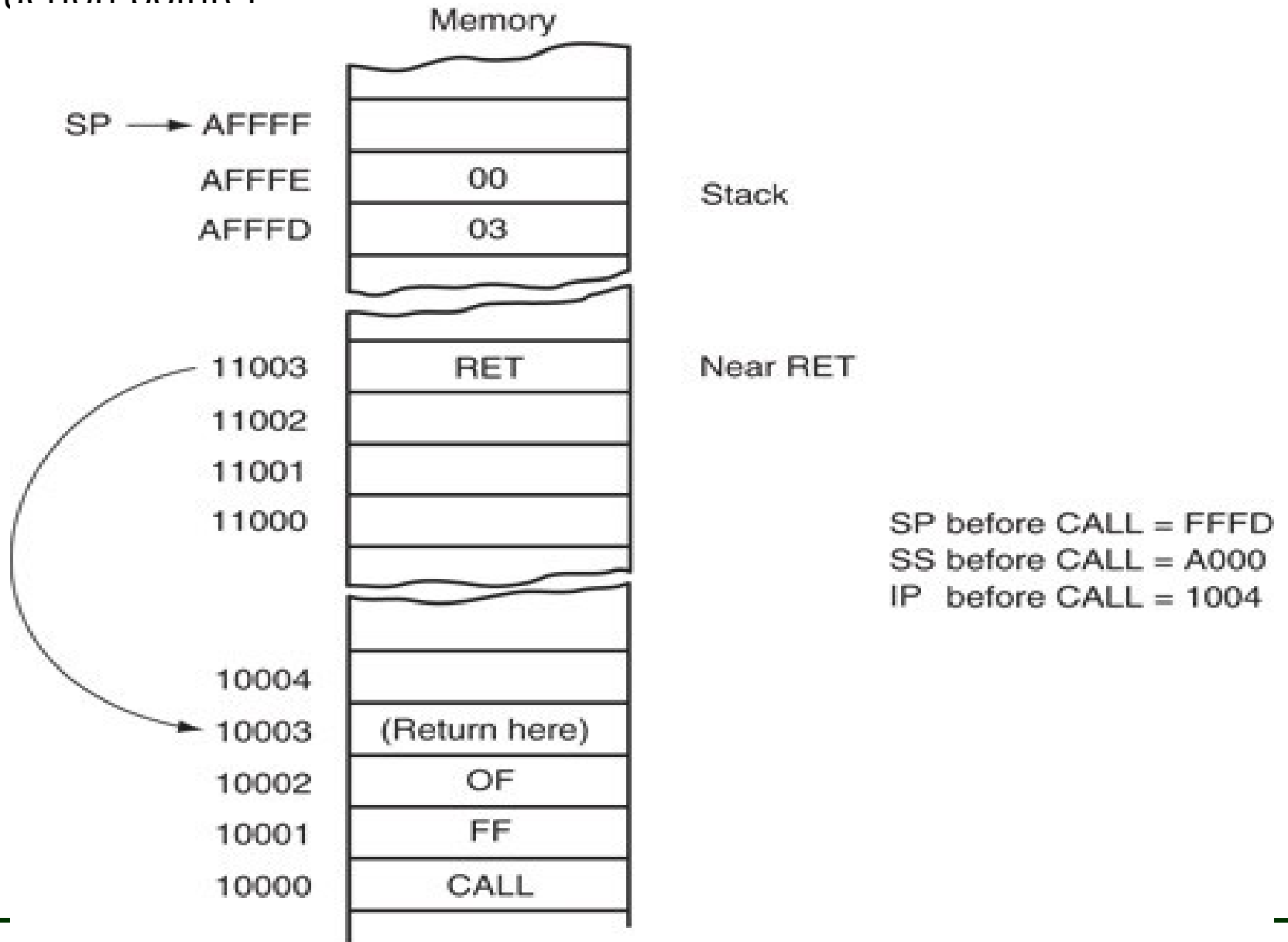
CALLs with Register Operands

- An example CALL BX, which pushes the contents of IP onto the stack.
 - then jumps to the offset address, located in register BX, in the current code segment
 - Always uses a 16-bit offset address, stored in any 16-bit register except segment registers.
-

RET

- Removes a 16-bit number (**near return**) from the stack placing it in IP, or removes a 32-bit number (**far return**) and places it in IP & CS.
 - near and far return instructions in procedure's PROC directive
 - automatically selects the proper return instruction
 - Figure 5-8 shows how the CALL instruction links to a procedure and how RET returns in the 8086-Core2 operating in the real mode.
-

Figure 5-8 The effect of a near return instruction on the stack and instruction pointer



- Another form of return adds a number to the contents of the stack pointer (SP) after the return address is removed from the stack.
 - A return that uses an immediate operand is ideal for use in a system that uses the C/C++ or PASCAL calling conventions.
 - these conventions push parameters on the stack before calling a procedure
 - If the parameters are discarded upon return, the return instruction contains the number of bytes pushed to the
-

- Parameters are addressed on the stack by using the BP register, which by default addresses the stack segment.
 - Parameter stacking is common in procedures written for C++ or PASCAL by using the C++ or PASCAL calling conventions.
-

INTRO TO INTERRUPTS

- An interrupt is a **hardware-generated CALL**
 - externally derived from a hardware signal
- Or a **software-generated CALL**
 - internally derived from the execution of an instruction or by some other internal event
 - at times an internal interrupt is called an *exception*
- Either type interrupts the program by calling
an **interrupt service procedure (ISP)**
or interrupt handler.

Interrupt Vectors

- A 4-byte number stored in the first 1024 bytes of memory (00000H–003FFH) in real mode.
 - in protected mode, the vector table is replaced by an interrupt descriptor table that uses 8-byte descriptors to describe each of the interrupts
 - 256 different interrupt vectors.
 - each vector contains the address of an interrupt service procedure
-

- Each vector contains a value for IP and CS that forms the address of the interrupt service procedure.
 - the first 2 bytes contain IP; the last 2 bytes CS
 - Intel reserves the first 32 interrupt vectors for the present and future products.
 - interrupt vectors (32–255) are available to users
 - Some reserved vectors are for errors that occur during the execution of software
-

- Some vectors are reserved for the coprocessor.
 - others occur for normal events in the system
 - In a personal computer, reserved vectors are used for system functions
 - Vectors 1–6, 7, 9, 16, and 17 function in the real mode and protected mode.
 - the remaining vectors function only in the protected mode
-

Interrupt Instructions

- Three different interrupt instructions available:
 - INT, INTO, and INT 3
- In real mode, each fetches a vector from the vector table, and then calls the procedure stored at the location addressed by the vector.
- In protected mode, each fetches an interrupt descriptor from the interrupt descriptor table.
- Similar to a far CALL instruction because it places the return address (IP/EIP and

INTs

- 256 different software interrupt instructions (INTs) available to the programmer.
 - each INT instruction has a numeric operand whose range is 0 to 255 (00H–FFH)
 - For example, INT 100 uses interrupt vector 100, which appears at memory address 190H–193H.
 - address of the interrupt vector is determined by multiplying the interrupt type number by 4
-

- Address of the interrupt vector is determined by multiplying the interrupt type number by 4.
 - INT 10H instruction calls the interrupt service procedure whose address is stored beginning at memory location 40H ($10H \times 4$) in the mode
 - In protected mode, the interrupt descriptor is located by multiplying the type number by 8
 - because each descriptor is 8 bytes long
 - Each INT instruction is 2 bytes long.
 - the first byte contains the opcode
 - the second byte contains the vector type
-

- When a software interrupt executes, it:
 - pushes the flags onto the stack
 - clears the T and I flag bits
 - pushes CS onto the stack
 - fetches the new value for CS from the interrupt vector
 - pushes IP/EIP onto the stack
 - fetches the new value for IP/EIP from the vector
 - jumps to the new location addressed by CS and IP/EIP
-

- INT performs as a far CALL
 - not only pushes CS & IP onto the stack, also pushes the flags onto the stack
 - The INT instruction performs the operation of a PUSHF, followed by a far CALL instruction.
 - Software interrupts are most commonly used to call system procedures because the address of the function need not be known.
 - The interrupts often control printers, video displays, and disk drives.
-

- INT replaces a far CALL that would otherwise be used to call a system function.
 - INT instruction is 2 bytes long, whereas the far CALL is 5 bytes long
 - Each time that the INT instruction replaces a far CALL, it saves 3 bytes of memory.
 - This can amount to a sizable saving if INT often appears in a program, as it does for system calls.
-

IRET/IRETD

- Used only with software or hardware interrupt service procedures.
 - IRET instruction will
 - pop stack data back into the IP
 - pop stack data back into CS
 - pop stack data back into the flag register
 - Accomplishes the same tasks as the far RET instruction followed by a POPF.
-

- When IRET executes, it restores the contents of I and T from the stack.
 - preserves the state of these flag bits
 - If interrupts were enabled before an interrupt service procedure, they are automatically re-enabled by the IRET instruction.
 - because it restores the flag register
 - IRET is used in real mode and IRETD in the protected mode.
-

INT 3

- A special software interrupt designed to function as a breakpoint.
 - a 1-byte instruction, while others are 2-byte
- Common to insert an INT 3 in software to interrupt or break the flow of the software.
 - function is called a breakpoint
 - breakpoints help to debug faulty software
- A breakpoint occurs for any software interrupt, but because INT 3 is 1 byte long, it is easier to use for this function.

INTO

- Interrupt on overflow (INTO) is a conditional software interrupt that tests overflow flag (O).
 - If $O = 0$, INTO performs no operation
 - if $O = 1$ and an INTO executes, an interrupt occurs via vector type number 4
 - The INTO instruction appears in software that adds or subtracts signed binary numbers.
 - with these operations, it is possible to have an overflow
-
- JO or INTO instructions detect the

Interrupt Control

- Two instructions control the INTR pin.
 - The **set interrupt flag** instruction (STI) places 1 in the I flag bit.
 - which enables the INTR pin
 - The **clear interrupt flag** instruction (CLI) places a 0 into the I flag bit.
 - which disables the INTR pin
 - The STI instruction enables INTR and the CLI instruction disables INTR.
-

- In software interrupt service procedure, hardware interrupts are enabled as one of the first steps.
 - accomplished by the STI instruction
 - Interrupts are enabled early because just about all of the I/O devices in the personal computer are interrupt-processed.
 - if interrupts are disabled too long, severe system problems result
-